EASYMOCK - EXPECTING CALLS

EasyMock provides a special check on the number of calls that can be made on a particular method. Suppose MathApplication should call the CalculatorService.serviceUsed method only once, then it should not be able to call CalculatorService.serviceUsed more than once.

```
//add the behavior of calc service to add two numbers and serviceUsed.
EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
calcService.serviceUsed();

//limit the method call to 1, no less and no more calls are allowed
EasyMock.expectLastCall().times(1);
```

Create CalculatorService interface as follows.

File: CalculatorService.java

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
   public void serviceUsed();
}
```

Example with calcService.serviceUsed called once

Step 1: Create an interface called CalculatorService to provide mathematical functions

File: CalculatorService.java

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
   public void serviceUsed();
}
```

Step 2: Create a JAVA class to represent MathApplication

File: MathApplication.java

```
public class MathApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      calcService.serviceUsed();
      return calcService.add(input1, input2);
   }

   public double subtract(double input1, double input2){
      return calcService.subtract(input1, input2);
   }

   public double multiply(double input1, double input2){
      return calcService.multiply(input1, input2);
   }
```

```
   public double divide(double input1, double input2){
      return calcService.divide(input1, input2);
   }
}
```

Step 3: Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by EasyMock.

File: MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   // @TestSubject annotation is used to identify class which is going to use the mock
object
   @TestSubject
   MathApplication mathApplication = new MathApplication();

   // @Mock annotation is used to create the mock object to be injected
   @Mock
   CalculatorService calcService;

   @Test
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1);

      //activate the mock
      EasyMock.replay(calcService);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test cases .

File: TestRunner.java

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(MathApplicationTester.class);
```

```
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Step 5: Verify the Result

Compile the classes using **javac** compiler as follows:

```
C:\EasyMock_WORKSPACE>javac Calculator Service.java Math Application.java Math
Application Tester.java Test Runner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

Example with calcService.serviceUsed Called Twice

Step 1: Create an interface CalculatorService to provide mathematical functions.

File: CalculatorService.java

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
   public void serviceUsed();
}
```

Step 2: Create a JAVA class to represent MathApplication.

File: MathApplication.java

```
public class MathApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      calcService.serviceUsed();
      calcService.serviceUsed();
      return calcService.add(input1, input2);
   }

   public double subtract(double input1, double input2){
      return calcService.subtract(input1, input2);
   }

   public double multiply(double input1, double input2){
      return calcService.multiply(input1, input2);
   }

   public double divide(double input1, double input2){
      return calcService.divide(input1, input2);
   }
}
```

Step 3: Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by EasyMock.

File: MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   // @TestSubject annotation is used to identify class which is going to use the mock
object
   @TestSubject
   MathApplication mathApplication = new MathApplication();

   //@Mock annotation is used to create the mock object to be injected
   @Mock
   CalculatorService calcService;

   @Test
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1);

      //activate the mock
      EasyMock.replay(calcService);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

Step 4: Execute test cases<

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE**to execute Test case*s*.

File: TestRunner.java

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(MathApplicationTester.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Step 5: Verify the Result

Compile the classes using **javac** compiler as follows:

```
C:\EasyMock_WORKSPACE>javac CalculatorService.java MathApplication.java
MathApplicationTester.java TestRunner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
testAdd(com.tutorialspoint.mock.MathApplicationTester):
   Unexpected method call CalculatorService.serviceUsed():
      CalculatorService.add(10.0, 20.0): expected: 1, actual: 0
      CalculatorService.serviceUsed(): expected: 1, actual: 2
false
```

Example without Calling calcService.serviceUsed

Step 1: Create an interface Calculator Service to provide mathematical functions

File: CalculatorService.java

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
   public void serviceUsed();
}
```

Step 2: Create a JAVA class to represent MathApplication

File: MathApplication.java

```
public class MathApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      return calcService.add(input1, input2);
   }

   public double subtract(double input1, double input2){
      return calcService.subtract(input1, input2);
   }

   public double multiply(double input1, double input2){
      return calcService.multiply(input1, input2);
   }

   public double divide(double input1, double input2){
      return calcService.divide(input1, input2);
   }
}
```

Step 3: Test the MathApplication class

Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be

created by EasyMock.

File: MathApplicationTester.java

```
import org.easymock.EasyMock;
import org.easymock.EasyMockRunner;
import org.easymock.Mock;
import org.easymock.TestSubject;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(EasyMockRunner.class)
public class MathApplicationTester {

   // @TestSubject annotation is used to identify class which is going to use the mock
object
   @TestSubject
   MathApplication mathApplication = new MathApplication();

   //@Mock annotation is used to create the mock object to be injected
   @Mock
   CalculatorService calcService;

   @Test
   public void testAdd(){

      //add the behavior of calc service to add two numbers
      EasyMock.expect(calcService.add(10.0,20.0)).andReturn(30.00);
      calcService.serviceUsed();
      EasyMock.expectLastCall().times(1);

      //activate the mock
      EasyMock.replay(calcService);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);

      //verify call to calcService is made or not
      EasyMock.verify(calcService);
   }
}
```

Step 4: Execute test cases

Create a java class file named TestRunner in **C:\> EasyMock_WORKSPACE** to execute Test cases .

File: TestRunner.java

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(MathApplicationTester.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Step 5: Verify the Result

Compile the classes using **javac** compiler as follows:

```
C:\EasyMock_WORKSPACE>javac Calculator Service.java Math Application.java Math
Application Tester.java Test Runner.java
```

Now run the Test Runner to see the result:

```
C:\EasyMock_WORKSPACE>java TestRunner
```

Verify the output.

```
testAdd(com.tutorialspoint.mock.MathApplicationTester):
   Expectation failure on verify:
      CalculatorService.serviceUsed(): expected: 1, actual: 0
false
```